Development of Reconfigurable Distributed Embedded Systems with a Model-Driven Approach

Fatma Krichen, Brahim Hamid, Bechir Zalila, Mohamed Jmaiel, Bernard Coulette

Development of reconfigurable distributed embedded systems with a model-driven approach

Fatma Krichen [1,*,†], Brahim Hamid [1], Bechir Zalila [2], Mohamed Jmaiel [2] and Bernard Coulette [1]

[1] *IRIT, University of Toulouse, France*
[2] *ReDCAD, National Engineering School of Sfax, University of Sfax, Tunisia*

SUMMARY

In this paper, we propose a model-driven approach allowing to build reconfigurable distributed real-time embedded (DRE) systems. The constant growth of the complexity and the required autonomy of embedded software systems management give the dynamic reconfiguration a big importance. New challenges to apply the dynamic reconfiguration at model level as well as runtime support level are required. In this direction, the development of reconfigurable DRE systems according to traditional processes is not applicable. New methods are required to build and to supply reconfigurable embedded software architectures. In this context, we propose an model-driven engineering based approach that enables to design reconfigurable DRE systems with execution framework support. This approach leads the designer to specify step by step his/her system from a model to another one more refined until the targeted model is reached. This targeted model is related to a specific platform leading to the generation of the most part of the system implementation. We also develop a new middleware that supports reconfigurable DRE systems.

KEY WORDS: software engineering; MDE-based approach; meta-model; UML profile; real-time embedded systems; non-functional properties; middleware

1. INTRODUCTION

An embedded system is composed of two main parts: software and hardware. The integration of these two parts achieves some functionalities. The dynamic reconfiguration consists in evolving the system from its current configuration to another configuration at runtime. Such system can evolve by either architectural or behavioral reconfigurations. The architectural reconfigurations consist in modifying the system topology such as adding or removing components or connections. The behavioral reconfigurations consist in modifying the system behavior by updating, for example, non-functional properties (NFPs) or implementations of components.

Most distributed real-time embedded (DRE) systems are not fully autonomous and require the human intervention to respond to events and to be reconfigured. But, human interventions can cause errors and require more time and much efforts. Moreover, it is sometimes impossible to interrupt a real-time critical system for reconfiguration. Therefore, the dynamic reconfiguration is required to construct autonomous DRE systems. Constructing reconfigurable DRE systems requires considerable efforts and is error prone. It is tedious and complex to develop these systems without providing models at high-level. New modeling concepts are required to specify dynamic reconfigurations of these systems. Moreover, the hardware resources of an embedded system are generally limited and

*Correspondence to: Fatma Krichen, IRIT, University of Toulouse, France 118 Route de Narbonne 31062 Toulouse Cedex 9, France.
†E-mail: fatma.krichen@irit.fr

their use has to be optimized. To develop a rich embedded system with several functionalities and low cost hardware resources, the hardware resources should be allocated only when required. For component-based architectures, components should be replaced/updated at runtime to be reused and to provide different functionalities.

Most of research activities present reconfigurable systems with a predefined number of configurations. Most achieved work in this direction have been proposed in the context of the two standards AADL (Architecture Analysis & Design Language) [1] and MARTE (Modeling and Analysis of Real-Time Embedded systems) [2]. Both standards define the dynamic reconfigurations in terms of modes and mode transitions. A mode represents a particular configuration while a transition represents an event, which applies the reconfiguration of system from a mode to another. The reconfigurations are described using state machines composed of a predefined number of modes and transitions between them. The major drawback of these two standards is the necessity of the definition of a predefined number of configurations.

Facing the exponential evolution of reconfigurable DRE system requirements, developers have very little time to market their systems. This constraint is an important factor to have a competitive advantage. For this reason, developers should build a system as fast as possible, guaranteeing required functionalities. To cope with the growing complexity of system design, several refinement approaches have been proposed. The most popular one is model-driven engineering (MDE) [3]. Using modeling languages in MDE, models represent the main artifacts to be constructed and maintained. In the MDE context, software development consists in transforming a model into another one more refined until the targeted model is reached. This targeted model is related to a specific platform and it is ready to be executed.

In this paper, we aim to provide an approach allowing to design reconfigurable DRE systems. For this, we propose an MDE-based approach that defines a set of steps to be followed by the developer. A set of transformation rules allow model to model transformations. Contrary to AADL [1] and MARTE profile [2], our approach allows specifying a reconfigurable system with a non-predefined number of configurations. For this, we introduce new concepts capturing the dynamic reconfigurations of DRE systems. Moreover, we have developed a new middleware that ensures the dynamic reconfiguration as well as the monitoring and the coherence of DRE systems.

The rest of the paper is organized as follows. In Section 2, we describe our whole development process to conceive reconfigurable DRE systems. Section 3 presents the proposed reconfigurable component architecture for real-time embedded system (RCA4RTES) model-based approach to specify reconfigurable DRE systems. Section 4 describes the proposed middleware dedicated to reconfigurable real-time embedded systems. The strategy of code generation is described in Section 5. Section 6 illustrates the effectiveness of the proposed approach by considering a case study having dynamic reconfiguration requirements: a global positioning system (GPS). In Section 7, we briefly review some related work that address the development process of embedded systems. Finally, Section 8 concludes this paper and presents some future work.

2. AN MODEL-DRIVEN ENGINEERING BASED APPROACH TO RECONFIGURABLE DRE SYSTEMS DEVELOPMENT

To solve the previously mentioned problems, we propose an MDE-based approach to design reconfigurable DRE systems with a non-predefined number of configurations. For that, we introduce the new concept *MetaMode* that captures and characterizes a set of configurations (modes) instead of defining each one of them. The *MetaMode* is described by structured components, connectors as well as non-functional and structural constraints. The modes belonging to a *MetaMode* are specified by the set of instances of structured components and connectors defined by this *MetaMode* and satisfying its constraints.

Our approach defines policy-based reconfigurations. We specify dynamic reconfigurations using state machines, which define a set of *MetaModes* and transitions between them. A transition between two *MetaMode* represents a set of reconfigurations between modes belonging to these *MetaModes* (as shown in Figure 1). When an event (represented as a *MetaMode* transition) is triggered, reconfigurations (i.e., represented as *mode* transition) are applied on the current mode to reach one of

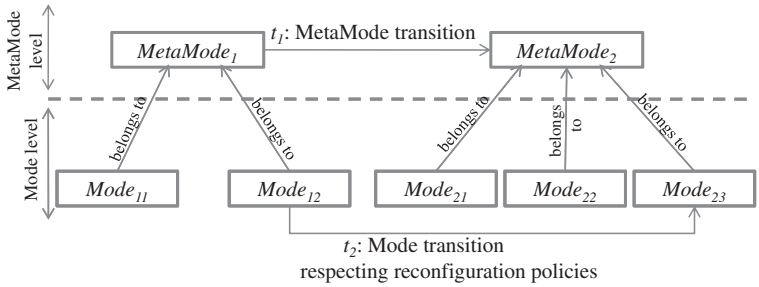

Figure 1. *MetaMode* modeling.

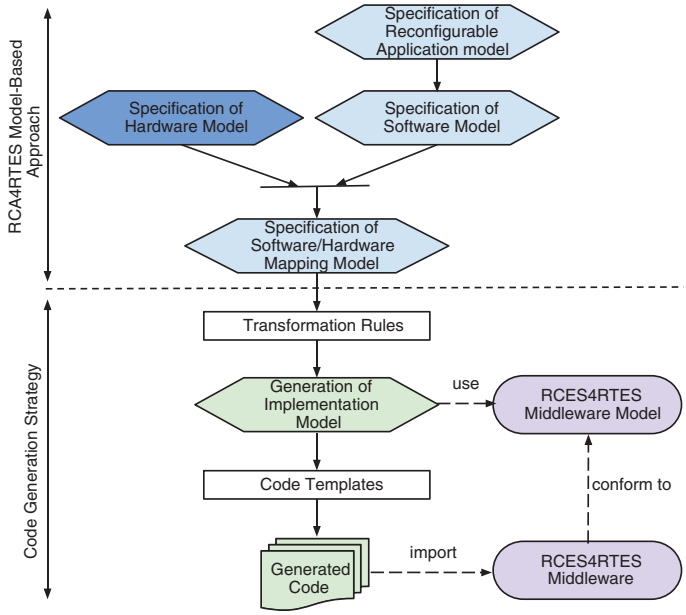

Figure 2. Development process for reconfigurable distributed real-time embedded systems.

the modes belonging to the target *MetaMode*. Reconfiguration policies allow to automatically select the target mode. The considered reconfiguration policies in this framework are memory, CPU and bandwidth optimization.

Figure 1 depicts the previous concepts with a toy example. The designer specifies the dynamic reconfigurations of his/her DRE system using a state machine, which contains two *MetaModes*: $MetaMode_1$ and $MetaMode_2$. A transition t_1 represents a reconfiguration between these two *MetaModes*. The mode transition t_2 is one of the possible transitions deduced from t_1 thanks to reconfiguration policies. The current mode $Mode_{12}$ is automatically replaced by the mode $Mode_{23}$.

Then, each *MetaMode* must be allocated on the hardware architecture. As the hardware architecture is unchanged, the allocation is defined from software architecture models (*MetaModes*) to execution supports. Some allocation constraints should be defined in order to specify the allocation policies. These policies define the mapping from software models to hardware instance.

All previously described concepts allow to model reconfigurable DRE systems. To perform the code generation, these models will be transformed to implementation models. The implementation models allow to generate code using the routines of our proposed reconfigurable component execution support for real-time embedded systems (RCES4RTES) middleware [4].

We define an MDE-based approach that defines a development process from models to code as shown in Figure 2. This process consists of five phases to be followed by user:

- **Specification of reconfigurable application model:** modeling of the dynamic reconfigurations using state machines composed of a set of *MetaModes* and transitions between them. Reconfiguration policies should be also specified to select the target mode. This model is conform to RCA4RTES meta-model [5].
- **Specification of software model:** modeling of the system *MetaModes* where each *MetaMode* is composed of a set of structured components, connectors as well as non-functional and structural constraints. Software model is also conform to RCA4RTES meta-model.
- **Specification of hardware model:** modeling of the fixed hardware architecture in terms of hardware components (such as processor and bus) using MARTE profile [2].
- **Specification of software/hardware mapping model:** allocation of system *MetaModes* to the specified fixed hardware architecture using RCA4RTES meta-model.
- **Automatic generation of implementation model:** generation of models that represent the system implementation. This model is conform to implementation meta-model [6]. From implementation model, a code is generated using the proposed RCES4RTES middleware [4].

3. RECONFIGURABLE COMPONENT ARCHITECTURE FOR REAL-TIME EMBEDDED SYSTEM MODEL-BASED APPROACH

In this section, we describe the new introduced concepts to specify reconfigurable DRE systems. These concepts allow to specify reconfigurable application model, software model and software/hardware mapping model as described in the previous section. For this, a new meta-model, called RCA4RTES, has been proposed to describe these new concepts and the relations between them. As implementation of this proposed meta-model, a Unified Modeling Language (UML) profile has been also proposed.

3.1. The reconfigurable component architecture for real-time embedded system meta-model

In the following, we detail the proposed RCA4RTES meta-model [5] shown in Figure 3 to specify reconfigurable DRE systems.

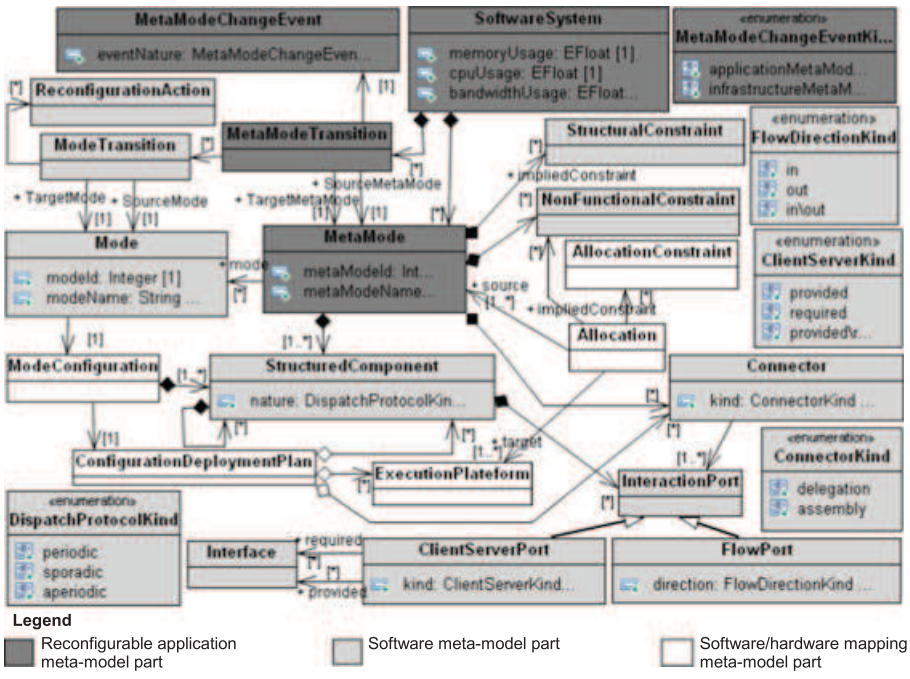

Figure 3. Reconfigurable component architecture for real-time embedded system meta-model.

As we describe the dynamic reconfigurations of these systems using state machines representing a set of *MetaModes* and transitions between them, we introduce the *SoftwareSystem* meta-class, which has a set of *MetaModes* and *MetaMode* transitions. A transition represented by the *MetaModeTransition* meta-class allows switching the system from a *MetaMode* to another when an event is triggered. In our approach, we handle two kind of events: an *application event* and an *infrastructure event* (*MetaModeChangeEventKind* enumeration). An application event represents a configuration change in accordance with user requirements while an infrastructure event represents a variation of situation in the infrastructure. To each transition, an activity of reconfiguration is associated. It represents an algorithm for switching from the current configuration (Mode) to the target one. A *MetaMode* transition represents a characterization of a set of mode transitions.

To specify the required reconfiguration policies, we introduce three properties (*cpuUsage*, *memoryUsage* and *bandwidthUsage*) for the *SoftwareSystem* meta-class. Each property should have a value to indicate the rate of consumption to not exceed by the corresponding resources.

To define the *MetaMode* that is composed of a set of structured components, connectors and structural and non-functional constraints, we introduce the *MetaMode* meta-class and the *StructuredComponent* meta-class that is composed of a set of interaction ports. Each structured component can be a periodic, a sporadic or an aperiodic thread (*DispatchProtocolKind* enumeration), or a composition of structured components. To describe the communication between components, we introduce the *Connector* meta-class that links two or more interaction ports. A connector can be a delegation connector (between two output ports or two input ports) or an assembly connector (between an input and output ports). In fact, we treat two kind of ports: flow port and client server port. A flow port presented by the *FlowPort* meta-class has been introduced to describe the data flow-oriented communication between components while a client server port that is presented by the *ClientServerPort* meta-class has been added to define a request/reply communication paradigm between components such as operation calls or signals.

Each *MetaMode* has several instances described using the *Mode* meta-class. For each mode, a configuration relates the mode to the deployment plan. A deployment plan describes a configuration by a set of structured components, the connections between them, their configuration and their allocation to physical nodes. We introduce the *Allocation* meta-class to specify the allocation of *MetaModes* to execution supports (e.g., allocation of software models to a fixed hardware architecture). This allocation has non-functional and allocation constraints that must be respected. In fact, our meta-model defines three kinds of constraints:

- Structural constraints are related to the topology of component-based architectures. For example, we can force the number of instances of `Receiver` component to be more than zero and less than four.

  $Receiver \rightarrow size{>}0$ *and* $Receiver \rightarrow size{<}4$
- Non-functional constraints specify conditions on the NFPs associated with models (i.e., components and connectors).

  For example, the processor frequency can be adjusted depending on the workload. If the processor utilization exceeds 90%, then the clock frequency is 60 MHz, else the clock frequency is 20 MHz.

  {procUtiliz > (90, percent) ? clockFreq==(60, MHz): clockFreq==(20, MHz)}
- Allocation constraints specify the policies used for the allocation of software models (*MetaModes*) to a fixed hardware architecture (i.e., execution supports). The allocation constraints are described using Value Specification Language (VSL) of MARTE [2]. Figure 4 shows an example of allocation constraint using VSL language. Each new instance of *SoftwareComponent* Component should be allocated on `cpu1` if component size is an even number or on `cpu2` otherwise.

3.2. The reconfigurable component architecture for real-time embedded system Unified Modeling Language profile

To handle reconfiguration requirements of DRE systems, a UML profile has been derived from the RCA4RTES meta-model. Figure 5 shows the dependencies of our profile. It imports both NFPs and

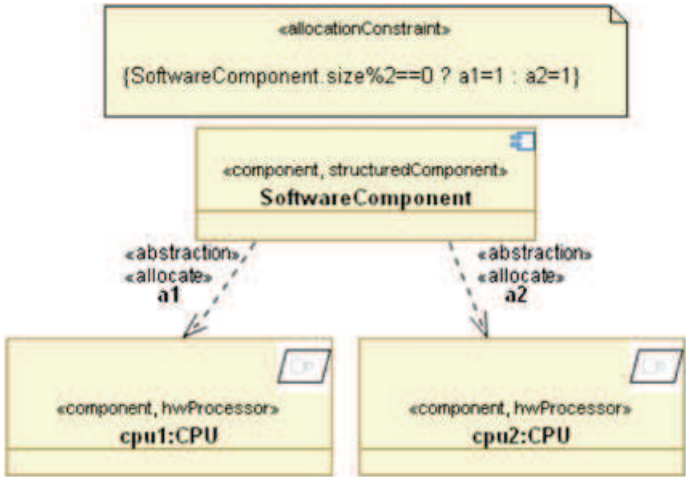

Figure 4. Example of allocation constraint.

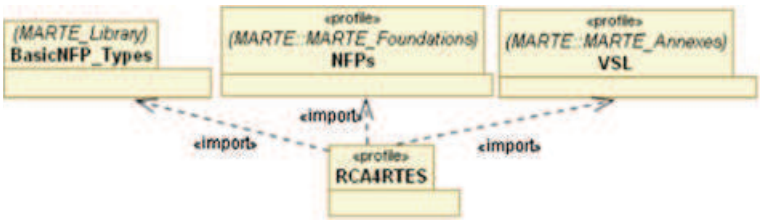

Figure 5. Reconfigurable component architecture for real-time embedded system (RCA4RTES) profile.

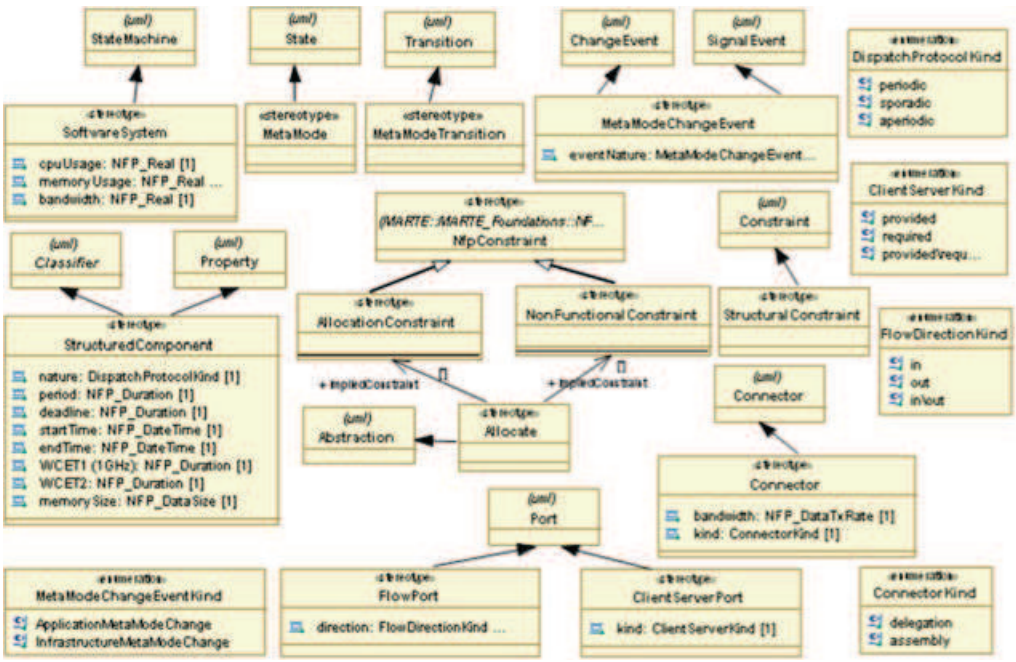

Figure 6. Reconfigurable component architecture for real-time embedded system profile description.

VSL profiles of MARTE [2] to specify non-functional and allocation constraints and the Basic NFP types of the MARTE library [2] to use the types defined in this library. The full profile description is given in Figure 6.

As mentioned previously, a *MetaMode* characterizes the system state by a set of structured components, connectors and structural and non-functional constraints. Therefore, the *MetaMode* stereotype extends the *State* UML meta-class. As we are interested in real-time embedded systems, each structured component is considered as a thread or a set of threads. For defining and characterizing these threads, we define the following properties as tagged values of *StructuredComponent* stereotype, which extends both *Classifier* and *Property* UML meta-classes:

- **Nature** defines the dispatch protocol of a component (periodic, sporadic or aperiodic thread).
- **Period** defines the period of a periodic thread. It is also used to describe the minimal time between two successive activations of a sporadic thread. Its type is *NFP_Duration* of MARTE.
- **Deadline** defines the deadline for periodic and sporadic threads. Its type is *NFP_Duration* of MARTE.
- **StartTime** defines the start time of an aperiodic thread. Its type is *NFP_DateTime* of MARTE.
- **EndTime** defines the end time of an aperiodic thread. Its type is *NFP_DateTime* of MARTE.
- **WCET** represents the Worst Case Execution Time computed as the sum of *WCET1* and *WCET2* of a thread:
  - *WCET1* defines the worst case execution time on a processor with 1 GHz of frequency. The *WCET1* on a processor with 1 MHz of frequency is computed by the ratio of the instruction number on the processor frequency. *WCET1* value varies according to the processor frequency. We can then compute this value depending on the processor frequency.
  - *WCET2* represents the time that does not depend on processor frequency but on other devices such as buses or memories.

The *Connector* stereotype that extends the *Connector* UML meta-class has the property *bandwidth* that is defined as tagged value to define the data rate of each connection between components. The type of this property is *NFP_DataTxRate* of MARTE library.

The *StructuralConstraint* stereotype extends the *Constraint* UML meta-class in order to specify architectural constraints using Object Constraint Language (OCL). Both *NonFunctionalConstraint* and *AllocationConstraint* stereotypes inherit from the *NfpConstraint* stereotype of NFP package of MARTE profile. These inheritances allow to use VSL [2], which is an extension of OCL, and allows to specify NFPs and constraints as well as the complex expressions of time.

The *MetaModeChangeEvent* stereotype, which extends both *SignalEvent* and *ChangeEvent* UML meta-classes is used to define the events that handle the system state machine. To ensure the allocation of *MetaModes* to execution supports, we define the *Allocate* stereotype, which extends the *Abstraction* UML meta-class. This stereotype is associated with non-functional and allocation constraints.

We also define a set of stereotypes to model the dynamic reconfigurations of DRE systems using state machines representing transitions between *MetaModes*. For this reason, we introduce the *SoftwareSystem* stereotype, which extends the *StateMachine* UML meta-class, and the *MetaModeTransition* stereotype, which extends the *Transition* UML meta-class.

To specify reconfiguration policies, we define three tagged values (*cpu usage*, *memory usage*, and *bandwidth usage*) into the *SoftwareSystem* stereotype which represent the maximum rate of consumption of resources. The type of these tagged values is *NFP_Real* of MARTE library.

4. PROPOSED RECONFIGURABLE COMPONENT EXECUTION SUPPORT FOR REAL-TIME EMBEDDED SYSTEM MIDDLEWARE

The RCES4RTES middleware [4] supports reconfigurable DRE systems. The central function of the proposed middleware is the dynamic reconfiguration of software component-based DRE systems. RCES4RTES also provides other functions required for developing reconfigurable real-time embedded applications. To develop the proposed middleware, we have extended the PolyORB_HI middleware [7]. Our middleware consists in

- Supporting the monitoring of the system by supervising at runtime the topology and the behavior of the architecture, tracing the system execution (e.g., obtaining the number of components and connections). The monitoring function can also be used to ensure the reflexivity of the system.
- Preserving the coherence of the system during and after reconfigurations since reconfiguration may lead the system to incoherent states.
- Respecting real-time constraints. In fact, on each node of the system, a *dynamic reconfiguration thread* is automatically created. It represents a sporadic thread applying reconfiguration actions. It will be considered as a system thread and then is scheduled with the other system threads. Using this sporadic thread, our middleware can easily manage the reconfigurations without affecting the system threads execution and exceeding thread deadlines.
- Ensuring the communication among heterogeneous platforms using the schizophrenic architecture and its canonical services. Taking advantage of these services and contrary to the existing middleware, the RCES4RTES middleware has a small memory footprint.
- Respecting the restrictions of Ravenscar profile [8] to ensure schedulability and both deadlock and livelock freedom of system threads.

Figure 7 shows a part of the RCES4RTES middleware model ensuring the dynamic reconfiguration and the communication between different threads. This middleware supports three kinds of threads: *PeriodicTask*, *SporadicTask* and *AperiodicTask* classes. These threads communicate using input and output ports defined by *Inport* and *OutPort* classes. This communication is ensured using *PortRouter* and *GeneratedType* classes. The *Event* class allows to specify system events that trigger reconfiguration actions. The trigging of an event changes the current system *MetaMode*. The *Mode* class is used by *ReconfigurationTrigger* class to specify the source mode and the target mode in order to identify the reconfigurations to apply. *ReconfigurationTrigger* class uses functions offered by the *ReconfDyn* class (*addTask*, *removeTask*, *addConnexion*, *removeConnexion*, etc.) to apply reconfigurations. Then, this class updates attributes of the *Context* class that represent the topology of the application at every time.

Further details about the RCES4RTES functions and their advantages are described in the following sub-sections.

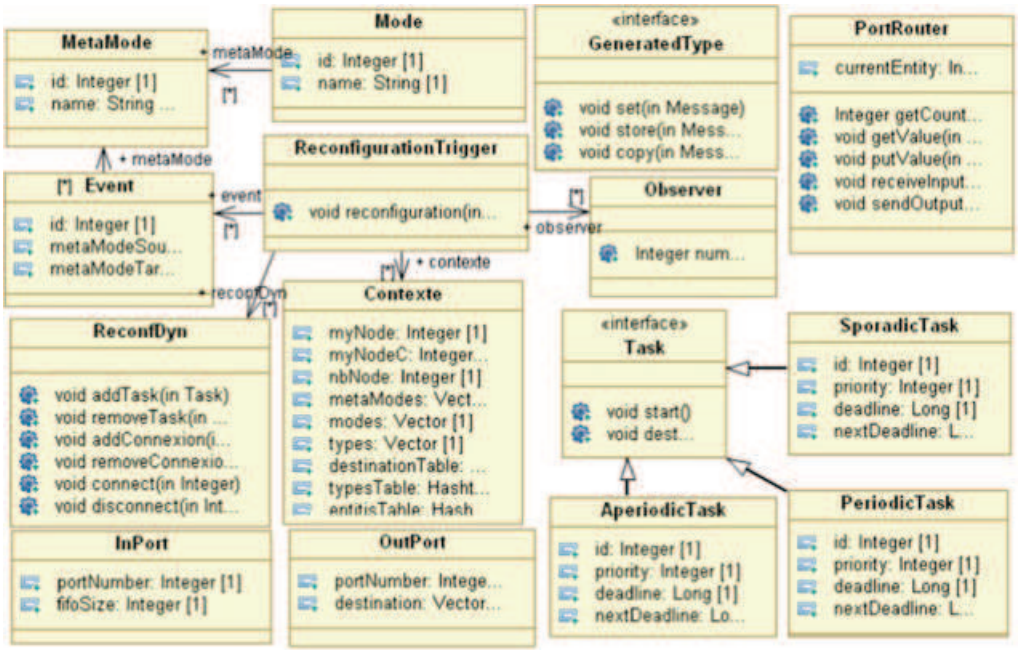

Figure 7. Part of the reconfigurable component architecture for real-time embedded system middleware model.

4.1. Dynamic reconfiguration

The RCES4RTES middleware performs the dynamic reconfiguration of DRE systems through two kinds of reconfigurations: architectural reconfigurations and behavioral reconfigurations. It handles the following architectural reconfigurations: *Connect nodes*, *Disconnect nodes*, *Add connection*, *Remove connection*, *Add component*, *Remove component* and *Migrate component*. It also handles the following behavioral reconfigurations: *Update component properties* and *Replace component*.

The RCES4RTES middleware introduces a set of data structures in the *Context* class to ensure the previous reconfigurations and to update the current state of the application in terms of nodes, components, connections and ports:

1. *myNodesC* data structure is defined in each node to describe the connections between the current node and the other nodes and to manage the dynamic interconnection of nodes. Each pair of nodes having at least one connection between their associated components should be connected. Connecting and disconnecting nodes require the update of the corresponding data structures.
2. *destinationTable* data structure is introduced to update at runtime the state of the interconnection between components. In each node, this data structure represents the destination ports of each deployed component port. Adding or removing connector between two ports (i.e., components) requires updating the corresponding data structures.
3. Both *entitiesTable* and *portsTable* data structures are defined in each node for adding, removing and migrating components at runtime. *EntitiesTable* data structure contains all application nodes with their corresponding deployed component instances while *portsTable* data structure contains all application component instances with their related ports. When a component has been added, removed or migrated, these two data structures are updated in each node to ensure the system coherence.

4.2. Coherence

The coherence is an essential property of a reconfigurable system. The system should be in a correct state during and after reconfigurations to prevent failures. To maintain the correct state of a system, we should avoid message loss between components during reconfigurations. Each component affected by the reconfiguration process should be locked during the reconfiguration. For this, we define two routines for locking and unlocking components. The locking of component consists in preventing the source components (i.e., the components that send requests to the locked component) to send requests and achieving the treatment of all current requests. The unlocking of component consists in releasing the lock by allowing the source components to send requests.

To avoid wasting time and to minimize as possible the locking duration, locking and unlocking components should be made respectively after creating new components and before deleting components in the reconfiguration routines.

4.3. Hard real-time

The time of reconfiguration T_{rd} (represented by the Equation (1)) is the sum of the locking duration of components T_b, the execution time of reconfiguration actions T_{act} and the transfer time of component state T_{state}. T_{state} is defined only in the case of the migration of component from a node to another while T_{act} is the execution time of the dynamic reconfiguration thread. For each node, a sporadic thread (i.e., dynamic reconfiguration thread) is created to perform the reconfigurations on this node and to inform the other nodes of these reconfigurations. This thread allows respecting time constraints (i.e., all system threads should meet their deadlines during and after reconfigurations). It will be considered and scheduled with system threads.

$$T_{rd} = T_b + T_{state} + T_{act} \tag{1}$$

T_b, T_{state} and T_{act} are proportional to the size of data to be treated. T_{act} also depends on the processor frequency and the transport layer. To obtain a deterministic reconfiguration time, we can so use a deterministic transport layer such as SpaceWire [9].

4.4. *Conformance to the Ravenscar profile*

The Ravenscar profile [8, 10] introduces restrictions allowing the schedulability and the deadlock and livelock freedom for real-time embedded systems. For efficiently performing the scheduling, the Ravenscar profile avoids the use of threads, which are randomly launched. It recommends the use of periodic and sporadic threads. Therefore, the set of threads to be analyzed is fixed and has static properties. Ravenscar profile also requires asynchronous communications. The communications between threads should be ensured only by a static set of protected shared objects, and the access to these protected shared objects should be carried out using Priority Ceiling Protocol [11].

In our approach, we are interested in three kinds of threads: periodic, sporadic and aperiodic threads. As each aperiodic thread has an arrival time, this thread respects the first Ravenscar profile restriction.

We also use asynchronous communications between system threads using Priority Ceiling Protocol. As we are focused on distributed systems, the time of both construction and sending of messages is non-deterministic because of the non-reliability of transport layers. This can be a source of message loss. To resolve this limitation, we propose to use a reliable and real-time transport layer such as SpaceWire [9]. We can therefore consider our distributed system as a local system.

The RCES4RTES middleware will be used to generate code of reconfigurable DRE systems.

5. IMPLEMENTATION AND CODE GENERATION STRATEGY

In order to generate the most part of a system implementation, a new meta-model and an implementation of this new meta-model have been also defined to describe the implementation model of each system. In the following sub-sections, we describe both implementation meta-model and profile, and we detail the code synthesis.

5.1. *Implementation meta-model*

The implementation meta-model described in Figure 8 allows the representation of system implementation models. As system implementations use routines defined in the RCES4RTES middleware, the implementation model imports the RCES4RTES middleware model.

Each distributed system implementation conform to our implementation meta-model has a set of processes. For this, we define both *System* and *Process* meta-classes. The system processes communicate to exchange data. The meta-class *Connector* describes the communication between

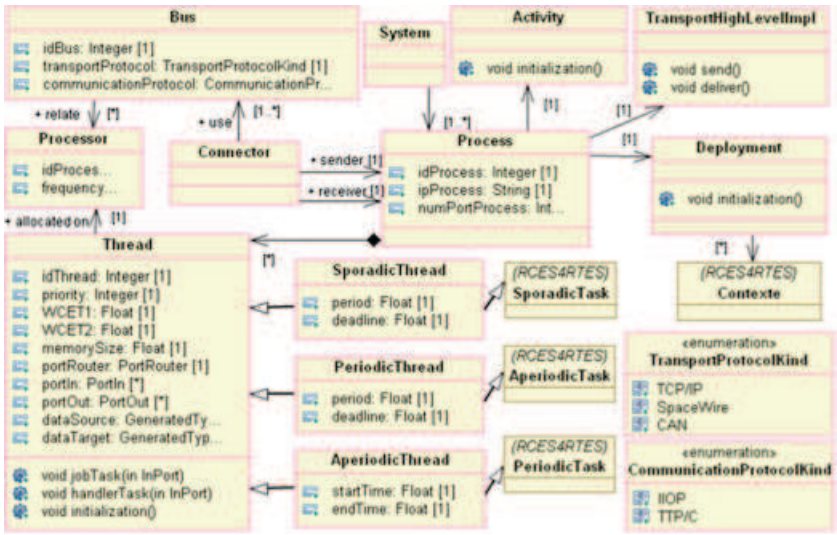

Figure 8. Implementation meta-model.

two processes (*sender* and *receiver*) through buses described by the *Bus* meta-class. Each bus is characterized by a communication and transport protocols (*CommunicationProtocolKind* and *TransportProtocolKind* enumerations).

A process is composed of a set of threads defined by the *Thread* meta-class. These threads can be periodic, sporadic or aperiodic threads defined respectively by *PeriodicThread*, *SporadicThread* or *AperiodicThread* meta-classes that inherit respectively from *PeriodicTask*, *SporadicTask* and *AperiodicTask* classes of RCES4RTES middleware model. Each thread is characterized by a set of NFPs such as priority and has a set of input and output ports whose types are respectively *PortIn* and *PortOut* classes of RCES4RTES middleware model. These ports allow to send and receive data whose type is *GeneratedType* class of RCES4RTES middleware model and through a port router (*PortRouter* class of RCES4RTES middleware model). The FP of each thread will be added by the developer in *threadJob* operation after the generation of code.

Each thread is allocated to a processor chosen according to the allocation constraints specified in the previous design level.

The *Processor* meta-class is characterized by a *Frequency* property. All processors are linked by buses defined by the *Bus* meta-class.

The following meta-classes are also defined: (i) *Deployment* meta-class representing the deployment of the initial mode using the *Context* class of RCES4RTES middleware model; (ii) *TransportHighLevelImpl* meta-class handling both sending and receiving data for each thread; and (iii) *Activity* meta-class allowing the starting of system threads and also the starting of the thread, which performs the dynamic reconfiguration of system (instance of *ReconfigurationTrigger* class of RCES4RTES middleware model).

5.2. Implementation profile

We propose a UML profile called Implementation profile (Figure 9) derived from our proposed implementation meta-model. We define both *System* and *Process* stereotypes, which extend the

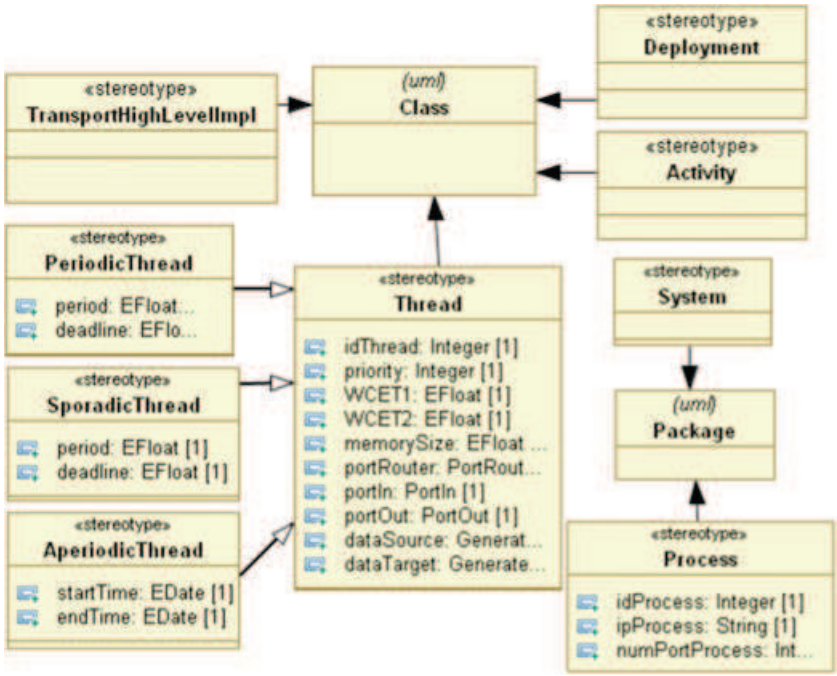

Figure 9. Implementation profile description.

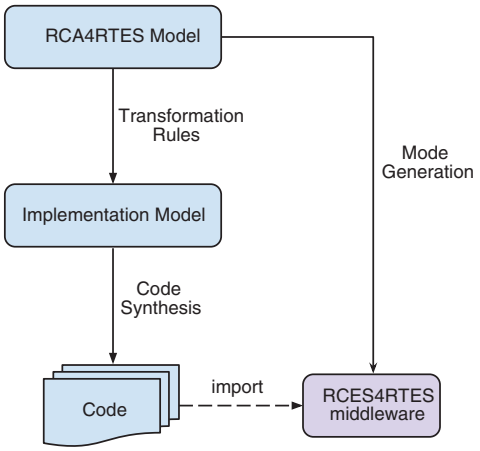

Figure 10. Code and mode generation.

Package UML meta-class to define the system as a set of packages. As defined in our implementation meta-model, each process is composed of a set of threads. For this, we define *PeriodicThread*, *SporadicThread* and *AperiodicThread* stereotypes. These stereotypes inherit from the *Thread* stereotype, which extends the *Class* UML meta-class. We also define *Deployment*, *TransportHighLevelImpl* and *Activity* stereotypes, which extend the *Class* UML meta-class.

5.3. Code synthesis

In our approach and as described in Figure 10, we provide a *code generator* allowing to generate code from RCA4RTES models. We also provide a *mode generator* allowing to enumerate the set of modes of each *MetaMode* compliant with the reconfiguration policies specified by the designer.

In order to generate code, the transformation from RCA4RTES model to implementation model is ensured by rules defined using Atlas Transformation Language [12]. For example, Listing 1 presents the transformation rule of UML state Machine where *SoftwareSystem* stereotype of RCA4RTES profile has been applied to UML package with *System* stereotype of implementation profile. Based on the developed RCES4RTES middleware, the code will be generated from the implementation model.

In order to generate modes, we have developed an algorithm allowing to select modes that are conform to reconfiguration policies specified in RCA4RTES models. This algorithm allows to directly add the selected modes to RCES4RTES middleware without passing through implementation model.

```
1  rule SoftSys2System {
2  from s : UML!StateMachine
3  (s.hasStereotype('SoftwareSystem'))
4  to   t : UML!Package (name <- s.name)
5  do {
6      -- Add the created package in the package of the model
7      thisModule.package.packagedElement <- t;
8      -- Apply the stereotype System on the generated package
9      t.applyStereotype(thisModule.getStereotype('System'));
10     }
11 }
```

Listing 1: Transformation rule of SoftwareSystem to System

5.4. Modeling framework tooling

A modeling framework has been proposed to conceive reconfigurable DRE systems. Figure 11 shows the used tools to make this framework. Our framework has been developed using the

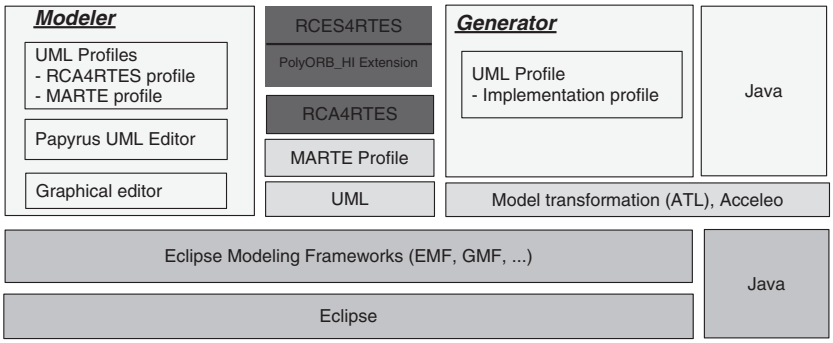

Figure 11. Modeling framework of reconfigurable distributed real-time embedded systems.

ECLIPSE[‡] platform. We use Papyrus[§] plug-in as a UML graphical editor. This editor is a UML editor where we can integrate UML profiles. In our proposed approach and in order to specify reconfigurable DRE systems, we integrate both RCA4RTES and MARTE profiles.

We use the Atlas Transformation Language[¶] language to define a set of transformation rules from RCA4ERTES model to implementation model, and the Acceleo[‖] project to define a set of patterns allowing to generate code from implementation model.

Our middleware has been implemented using Real-time Specification for Java. It represents an extension to the existing PolyORB_HI middleware. We have extended and updated this middleware to add routines allowing to support the dynamic reconfigurations as well as the coherence and the monitoring of DRE systems.

6. APPLICATION OF RECONFIGURABLE COMPONENT ARCHITECTURE FOR REAL-TIME EMBEDDED SYSTEM TO A GPS CASE STUDY

We illustrate our MDE-based approach using GPS [5] case study. A GPS is a radio navigation system that provides accurate navigation signals to any place on Earth. It helps the user to determine the road to be followed from his/her current place to some specified destinations using information provided by a satellite. The satellite sends to Earth an encrypted signal that contains various information useful for localization and synchronization. The control base sends and receives information to satellites in order to synchronize the clocks of satellites. We only define the following use cases: (i) GPS with insecure functioning: consists of a traditional (or public) use of a GPS; and (ii) GPS with secure functioning: represents a restricted use of a GPS with some safety requirements.

Following our development process described in Section 2, we begin by defining a state machine specifying the dynamic reconfigurations by a set of *MetaModes* and *MetaMode* transitions. We have defined a UML state machine diagram as shown in Figure 12. We specify two *MetaModes* of GPS: (i) *Insecure GPS MetaMode*; and (ii) *Secure GPS MetaMode*. The transition from one *MetaMode* to another is ensured by event triggering. For example, the switch from *Insecure GPS MetaMode* to *Secure GPS MetaMode* occurs when the monitor commands to move to the secure state.

The reconfiguration policies are also defined as tagged values of *SoftwareSystem* stereotype (Figure 12). For example, the memory consumption should not exceed 40% of hardware memory size. The modes conform to the specified reconfiguration policies are then generated and added to our middleware.

In the second step, each *MetaMode* should be described including structured components, connectors as well as non-functional and structural constraints. For the sake of simplicity, many functionalities of this case study have been omitted. Both satellite and control base are represented by basic

[‡]http://www.eclipse.org
[§]http://www.eclipse.org/modeling/mdt/papyrus/
[¶]http://www.eclipse.org/atl/
[‖]http://www.eclipse.org/acceleo

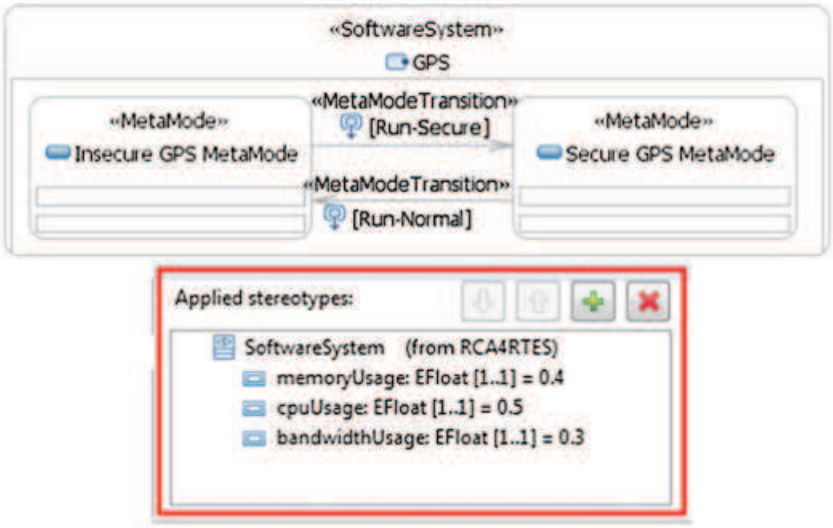

Figure 12. The state machine of GPS.

Table I. The non-functional properties of structured components of the GPS system.

Structured component	Nature	Period deadline (ms)	WCET1 (ms)	WCET2 (ms)	Memory size (MB)
Receiver	Sporadic	100	20	2	0.9
Position	Sporadic	100	20	2	0.5
TreatmentUnit	Sporadic	100	20	4	0.75
Decoder	Sporadic	100	20	2	0.1
Encoder	Sporadic	100	20	0	0.5
GpsSatellite	Periodic	400	30	0	0.9
GpsControlBase	Periodic	400	30	0	0.9

components (resp. *GpsSatellite* and *GpsControlBase* components). In this paper, we only describe the *GPS_Terminal* architecture that consists of five components for *Insecure MetaMode*:

- *Position* component for receiving the satellite signal.
- *Receiver* component for converting the analog signal into a digital signal.
- *Decoder* component for decoding digital information and separating between the information to calculate distance and time information.
- *TreatmentUnit* component for computing the distance from the satellite in order to obtain the position.
- *Encoder* component for encoding time and position information.

The switch from *Insecure GPS MetaMode* to *Secure GPS MetaMode* consists in removing all instances of *Position* component and adding instances of both *SecurePosition* and *AccessController* components to assure the secure reception and the satellite signal control. Table I presents the properties of components, which have been obtained using a simple simulation.

Then and in the third step, we specify the fixed hardware architecture followed by the allocation of each *MetaMode* to the specified fixed hardware architecture. Figure 13 presents the allocation of *Insecure GPS MetaMode* to GPS terminal hardware and GPS satellite hardware. The top part of Figure 13 describes the *Insecure GPS MetaMode* while the lower part shows the hardware architecture of both GPS terminal and GPS satellite. We use the MARTE profile to specify the hardware architecture. The allocation constraints describe the policies of allocation of models to hardware instances. For example, the allocation of instances of `Encoder` structured component is devised between the two processors `cpu1` and `cpu2` of GPS terminal.

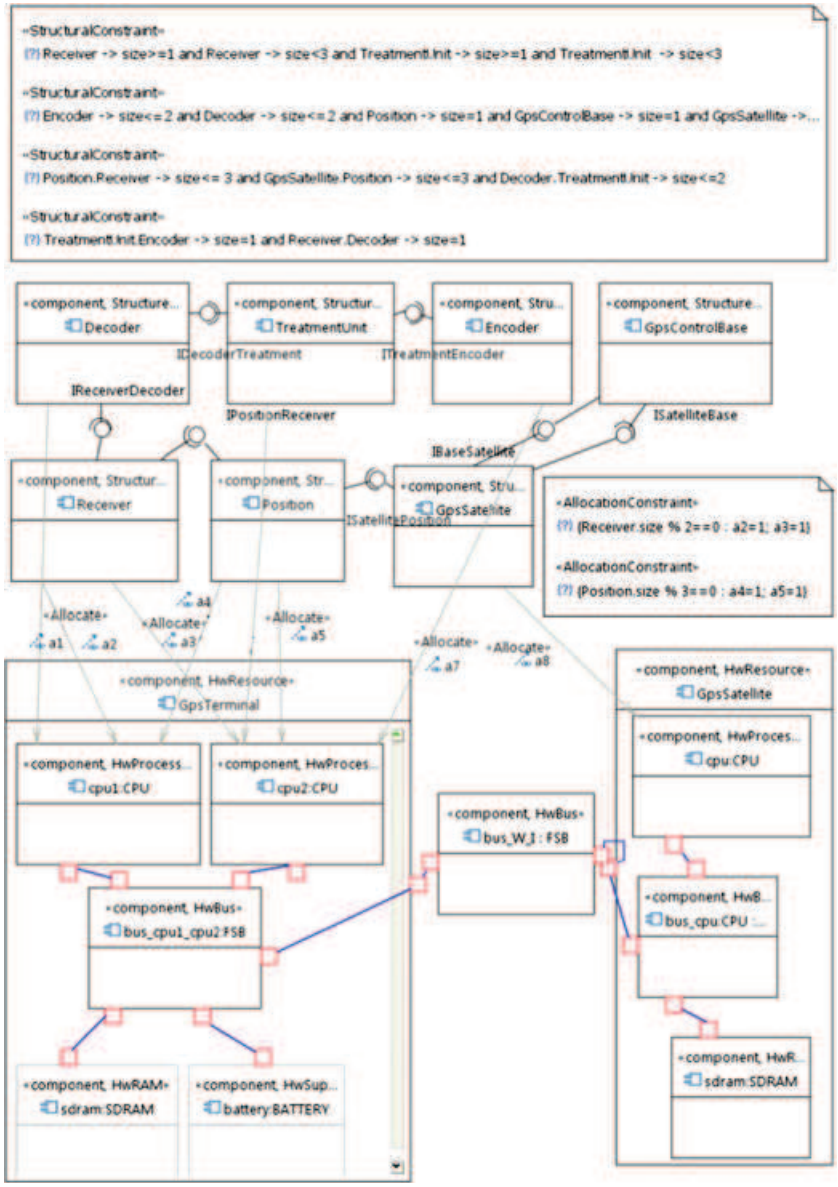

Figure 13. Allocation of insecure *MetaMode* to GPS terminal hardware and GPS satellite hardware.

Then and in order to generate code, the implementation model will be obtained by applying transformation rules. Figure 14 presents the GPS implementation model. The GPS_terminal implementation has seven classes representing sporadic threads (Receiver, Decode, Encode, etc.). The Activity class creates and launches instances of theses threads. The Deployment class represents the initial topology of system. The *TransportHighLevelImpl* class is generated to ensure the communication between threads.

After generating code and to illustrate the use of our middleware, we define two configurations of GPS: *Insecure GPS Configuration* (Figure 15) of *Insecure GPS MetaMode* as initial configuration and *Secure GPS Configuration* (Figure 16) of *Secure GPS MetaMode*.

Our middleware allows the dynamic reconfiguration by switching the GPS from *Insecure GPS Configuration* to *Secure GPS Configuration* when event is launched. This switching consists in removing all instances of Position component and adding instances of both *SecurePosition* and *AccessController* components to assure the secure reception and the satellite signal control. For

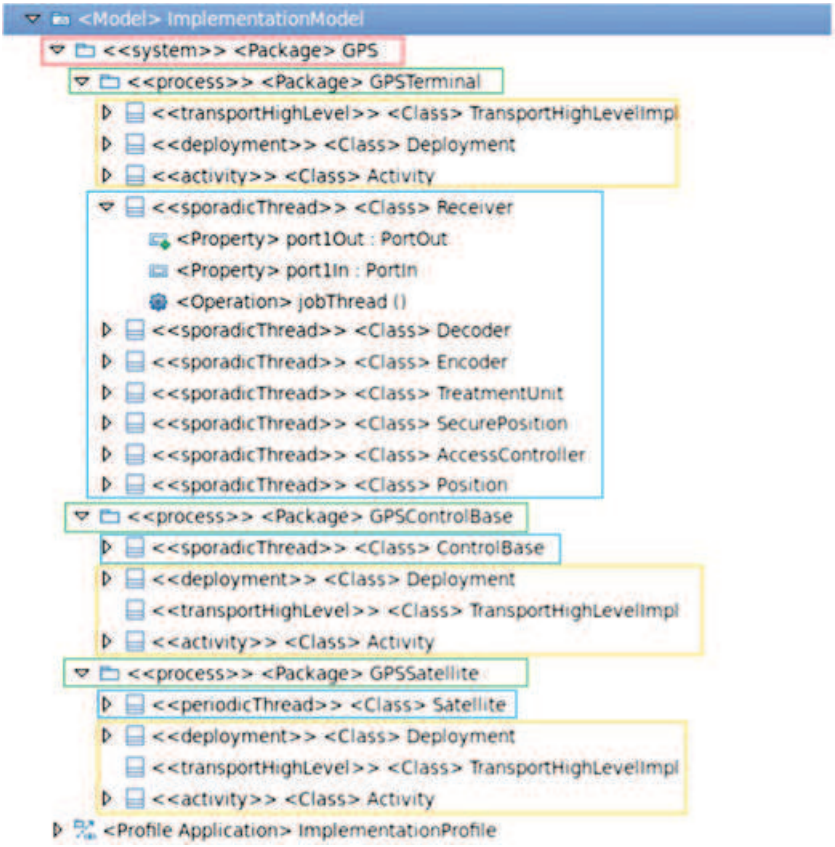

Figure 14. GPS implementation model.

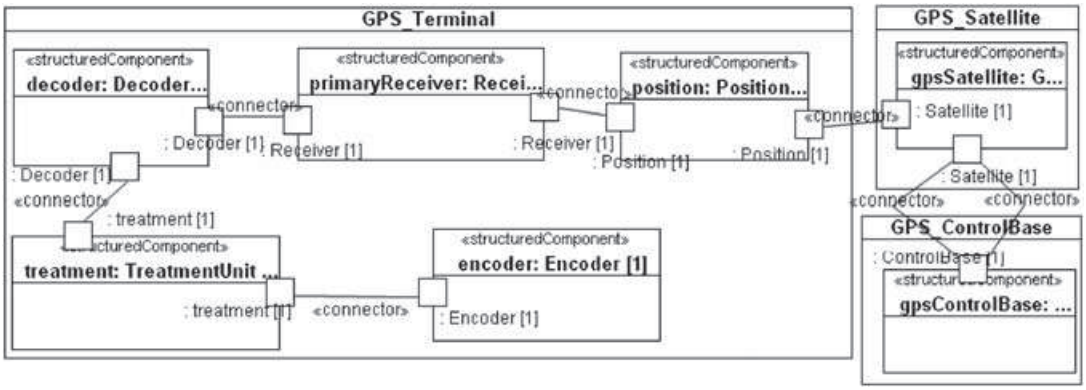

Figure 15. Insecure GPS configuration.

this, the connections of *position* instance with *gpsSatellite* and *primaryReceiver* instances should be removed and two instances of both *SecurePosition* and *AccessController* with their connections should be added as shown in Figure 17. A second instance of *Receiver* component should also be added.

After applying efficiently reconfigurations, we demonstrate also that the monitoring and the coherence are ensured by our middleware using the considered GPS case study. After the transition from *Insecure GPS Configuration* to *Secure GPS Configuration*, the system remains coherent and preserves its temporal constraints. As shown in Figure 18, we observe that both *primaryReceiver*

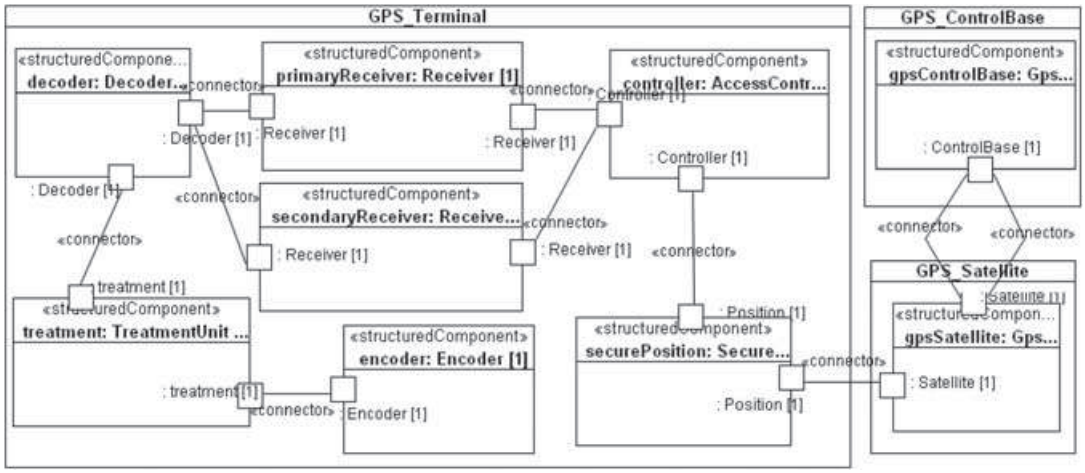

Figure 16. Secure GPS configuration.

```
Dynamic reconfiguration: the securePosition whose type is SecurePosition has been added
on GPS_Terminal node
Dynamic reconfiguration: the controller whose type is AccessController has been added
on GPS_Terminal node
Dynamic reconfiguration: the secondaryReceiver whose type is Receiver has been added
on GPS_Terminal node
GPS_Terminal node: send 18 bytes to GPS_Satellite node
Dynamic reconfiguration: distant connection between the port 17 and 11 has been added
Dynamic reconfiguration: local connection between the port 10 and 13 has been added
Dynamic reconfiguration: local connection between the port 12 and 3 has been added
Dynamic reconfiguration: local connection between the port 12 and 15 has been added
Dynamic reconfiguration: local connection between the port 14 and 5 has been added
Creation of the securePosition instance on GPS_Terminal node
securePosition : wait initialization
securePosition : new Dispatch
securePosition is waiting for incoming events
Creation of the controller instance on GPS_Terminal node
Creation of the secondaryReceiver instance GPS_Terminal node
Coherence: the position has been locked
controller  : wait initialization
Consistency: the position instance has been unlocked
secondaryReceiver : wait initialization
Dynamic reconfiguration: local connection between the port 0 and 3 has been removed
controller  : new Dispatch
secondaryReceiver : new Dispatch
isconnection : existe=true
controller  is waiting for incoming events
secondaryReceiver is waiting for incoming events
GPS_Terminal node : send 18 bytes to GPS_Satellite node
Dynamic reconfiguration: distant connection between the port 17 and 1 has been removed
Dynamic reconfiguration: position has been removed from the GPS_Terminal node
```

Figure 17. Log of the dynamic reconfiguration from *Insecure GPS Configuration* to *Secure GPS Configuration*.

and *secondaryReceiver* instances run normally and meet their deadlines (100 ms). For example, *secondaryReceiver* exits in about 57 ms.

In addition to the small memory footprint of our middleware ($\simeq$ 131 KB), we have computed the memory footprint of the GPS case study on each system node: 51.5 KB for the *GPS_Terminal* node, 25.6 KB for the *GPS_Satellite* node and 25.9 KB for the *GPS_ControlBase* node. So, we can conclude that the memory footprint for each node is small.

```
secondaryReceiver calls a sub program at (1321967893590 ms, 445144 ns)
primaryReceiver, getValue of input port 3
primaryReceiver, readIn : reading the oldest element in the queue of event [data] input port 3
primaryReceiver, readIn : value read from input port 3
primaryReceiver, readIn : reading the oldest element in the queue of event [data] input port 3
primaryReceiver, readIn : value read from input port 3
primaryReceiver, getValue done
primaryReceiver, nextValue for event [data] input port 3
primaryReceiver, dequeue : dequeuing event [data] input port 3
primaryReceiver, historyIncrementFirst : globalHistoryFirst = 4
primaryReceiver calls a sub program at (1321967893642 ms, 430784 ns)
secondaryReceiver  termine impl at (1321967893647 ms, 293352 ns)
 at t=14h18m13s647ms ***** the node 0 converted the signal into the value : 15
secondaryReceiver, storeOut : storing value for output port 14
secondaryReceiver, storeOut : value stored for output port 14
secondaryReceiver, sendOutput for output port 14
secondaryReceiver, setInvalid : setting invalid for output port 14
secondaryReceiver, sendOutput output port 14
secondaryReceiver, send output to input port 5 of entity 2
decoder : store received message
.....
decoder calls a sub program at (1321967893648 ms, 987482 ns)
primaryReceiver termine impl at (1321967893716 ms, 950898 ns)
```

Figure 18. Meeting of thread deadlines.

7. RELATED WORK

In this section, we review some related works that address the development of real-time embedded systems from models to execution platforms. A detailed state of the art has been presented in [13].

Several activities have been carried out to design embedded systems and particularly reconfigurable ones.

Architecture Analysis & Design Language [1] is an architecture description language, which allows the specification of DRE systems as a component assembly. It allows to describe both software and hardware parts of a system. AADL also allows to specify reconfigurable systems using state machines composed of modes and mode transitions. A mode represents a particular state (configuration) while a transition represents an event, which allows system reconfiguration. Compared with our approach, the modes in AADL are statically predefined. This considerably reduces the modeling possibilities. AADL specifies embedded systems at a low level (thread, processor, etc), so that the modeling of reconfigurations is related to a specific application and platform.

Modeling and Analysis of Real-Time Embedded systems [2] is a UML profile for MARTE systems inspired from the Scheduling, Performance and Time profile [14]. It allows the separation of both hardware and software parts of platform resources and the modeling of NFPs. It presents a set of packages that allow to specify a system at several levels of abstraction. Moreover, MARTE allows to specify the behavioral reconfigurations of real-time embedded systems using state machines composed of a set of modes and transitions between them. Contrary to our approach, MARTE does not support distributed systems. It allows to specify only the behavioral reconfigurations of system. It does not handle the architectural reconfigurations.

To cope with the growing complexity of embedded system design, several development processes have been proposed.

Ocarina is a framework that allows developing, configuring and deploying DRE systems using a model-driven approach [15, 16]. Using Ocarina, DRE systems can be specified using AADL. From AADL model, Ocarina can perform scheduling and verification analysis to ensure the validity of the model. Then, an important part of the application code as well as a middleware layer devoted to specific needs of the application will be generated. Ocarina allows automatic code generation from AADL models to multiple execution supports. Both deployment and configuration of DRE application are performed automatically by Ocarina using information extracted from AADL models. However, this framework does not address reconfiguration in DRE systems. It uses AADL language

that does not allows to specify DRE systems at a high level of abstraction. AADL specifies embedded systems at a low level (thread, processor, etc) and this requires more competence to specify these systems.

In the same direction, an MDA approach to address real-time software reusability, maintainability and portability issues is proposed in [17]. In fact, authors propose a model-driven framework that defines a new methodology that makes easier the design and implementation of real-time embedded systems. First, the application model should be annotated with High Level Application Modeling sub-profile of MARTE [2]. The target platform model and the mapping model should also be specified. Then, the generated platform specific model is obtained through defined generic transformation rules. Finally, the executable code is generated. As an outcome of this process, designers can obtain a real-time embedded system architecture that can be used for several platform implementations. However, this approach does not take into consideration the reconfiguration of such system.

ModES[18] is also an MDE-based approach devoted to embedded system design. It defines a set of meta-model representing the following: (i) application to capture functionality by means of processes communicating; (ii) platform to indicate available hardware/software resources; (iii) mappings from application to platform; (iv) and implementations, oriented to code generation and hardware synthesis. The particularity in this approach is that the mapping meta-model does not specify only the allocation of application processes to fixed hardware components. This mapping also delimits a design space that corresponds to all possible implementations that can be obtained through the choice of sequences of transformations between models. Therefore, the set of transformations between models implements the possible mappings from application to platform. The ModES methodology includes a set of tools that support model-based design tasks starting from specification until software/hardware generation and synthesis. However, this approach does not support the reconfiguration of DRE systems.

The previously presented approaches [17–19] offer development processes that allow to conceive real-time embedded systems. They present methodologies to be followed by developer from high level models to code. However, these approaches do not support reconfigurable systems. In this direction, TimeAdapt [20] is a development process for reconfigurable system design. It follows a three-tiered approach providing means to specify reconfiguration actions, estimate whether their execution can be carried out within a given time bound and execute them in a timely manner. In fact, each reconfiguration has time bound that is based on environmental conditions and structural application. An admittance test calculates the probability whether the given reconfiguration can meet the specified time bounds. If this probability exceeds a given threshold, the reconfiguration is scheduled as a high priority real-time task and its reconfiguration actions will be executed. In case of a reconfiguration task rejection, the reconfiguration is rescheduled with a new time bound at some later point in time. TimeAdapt supports the execution of reconfigurations on component-based real-time applications. However, this framework provides a bounded time for each reconfiguration. If a reconfiguration exceeds its estimated time, it will not be executed.

In the same context, COMponent-based Design of Embedded Software for Distributed Systems (COMDES) [21] is a framework dedicated to the specification and the configuration of real-time embedded systems. Using this framework, an embedded application is built from reusable components implemented as executable function blocks. COMDES defines a development process for embedded systems starting from design level until production of application code. A system is modeled in a high level of abstraction, and then the output model will be transformed into a COMDES model that will be generally enriched with information that guide code generation. Finally, the generated code is deployed and tested. This framework defines two types of processes: configuration process and reconfiguration process. The configuration process allows to find components in the component repository and then to assemble them to configure an application model. A reconfiguration process allows adding, removing and updating components at runtime in order to update the application. However, using COMDES framework, the developer has a limited number of prebuilt components that are stored into a component directory, then the developer can not add a new component that does not exist in the directory.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an MDE-based approach to design reconfigurable DRE systems. We developed a modeling framework that introduces new concepts to design reconfigurable DRE systems using meta-modeling technologies and following an MDE process. This process allows to easily design and generate an important part of DRE system implementation.

We specify reconfigurable DRE systems without enumerating all configurations by introducing the new concept *MetaMode*. The reconfigurations are specified using state machines having a set of *MetaModes* and transitions between them. We also ensure the allocation of each MetaMode to hardware architecture. We developed a new middleware as an extension and an updating of the PolyORB_HI middleware providing a set of routines performing the dynamic reconfigurations and ensuring the coherence and the monitoring. Then, we studied code generation targeting such a middleware. An implementation model is obtained thanks to transformation rules and then a code is generated.

Providing a high-level description in the process of construction of DRE systems minimizes errors and developer's efforts. Our modeling framework allows to specify autonomous systems where reconfigurations will be performed without human intervention. However, our approach does not assure the syntactic and semantic validation of specified models. Moreover, our approach support real-time embedded systems, but it does not support critical systems. We should add temporal constraints that must be respected to switch from a *MetaMode* to another and ensure the correct execution of these systems.

As future work, we aim to propose patterns that allow to easily specify reconfigurable DRE systems. Then, these patterns will be translated to models conforming to our RCA4RTES meta-model. In addition, we aim at enriching our modeling framework and to add new routines to our middleware to support the fault tolerance. We also aim to specify reconfigurable DRE systems using formal formalisms. In fact, we plan to migrate our modeling concepts to a formal language such as Z to ensure syntactic and semantic validation.

REFERENCES

1. SAE. Architecture Analysis & Design Language (AADL), January 2009.
2. OMG. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, 2009.
3. Schmidt DC. Guest editor's introduction: model-driven engineering. *Computer* 2006; **39**(2):25–31.
4. Krichen F, Zalila B, Jmaiel M, Hamid B. A middleware for reconfigurable distributed real-time embedded systems. In *Proceedings of the acis international conference on software engineering research, management and applications sera (selected papers)*, Studies in Computational Intelligence. Springer: Shanghai, China, 2012; 81–96.
5. Krichen F, Hamid B, Zalila B, Jmaiel M. Towards a model-based approach for reconfigurable distributed real time embedded systems. In *Proceedings of the 5th European Conference on Software Architecture*. Springer: Essen, Germany, 2011 September; 295–302.
6. Krichen F, Ghorbel A, Zalila B, Hamid B. An mde-based approach for reconfigurable dre systems. In *Proceedings of the 21st IEEE International Conference on Collaboration Technologies and Infrastructures*. IEEE Computer Society: Toulouse, France, 2012juin; 78–83.
7. Zalila B, Pautet L, Hugues J. Towards Automatic Middleware Generation. In *Proceedings of the International Symposium on Object-oriented Real-time distributed Computing*. IEEE, 2008; 221–228.
8. Burns A. The ravenscar profile. *Ada Letters* December 1999; **XIX**:49–52.
9. ECSS-E-ST-50-12C. Spacewire - links, nodes, routers and networks, European Space Agency July 2008. Technical report.
10. Kwon J, Wellings A, King S. Ravenscar-Java: a high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on java grande*. ACM: New York, NY, USA, 2002; 131–140.
11. Lui Sha RR, Lehoczky JP. Priority inheritance protocols : an approach to real-time synchronization. *IEEE transactions on computers* September 1990; **39**(9):1175–1185.
12. Jouault F, Allilaire F, Bézivin J, Kurtev I, Valduriez P. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006; 719–720.
13. Krichen F. Position paper: Advances in Reconfigurable Distributed Real Time Embedded Systems. In *International workshop on Distributed Architecture modeling for Novel component based Embedded systems*. IEEE: Tozeur, Tunisia, 2010; 273–278.
14. OMG. UML Profile for Schedulability, Performance, and Time Specification, January 2005.

15. Hugues J, Zalila B, Pautet L, Kordon F. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.* 2008; **7**(4):42:1–42:25.
16. Lasnier G, Zalila B, Pautet L, Hugues J. Ocarina : an environment for aadl models analysis and automatic code generation for high integrity applications. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*. Springer: Brest, France, 2009; 237–250.
17. Chehade WEH, Radermacher A, Terrier F, Selic B, Gérard S. A model-driven framework for the development of portable real-time embedded systems. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society: Las Vegas, Nevada, USA, 2011; 45–54.
18. do Nascimento FAM, Oliveira MFS, Wagner FR. Modes: embedded systems design methodology and tools based on mde. In *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE Computer Society: Washington, DC, USA, 2007; 67–76.
19. Zalila B. Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture. *Ph.D. Thesis*, École Nationale Supérieure des Télécommunications, 2008.
20. Fritsch S, Clarke S. Timeadapt: timely execution of dynamic software reconfigurations. In *Proceedings of the 5th Middleware doctoral symposium*. ACM: New York, NY, USA, 2008; 13–18.
21. Guo Y, Sierszecki K, Angelov C. A (re)configuration mechanism for resource-constrained embedded systems. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE Computer Society: Washington, DC, USA, 2008; 1315–1320.